

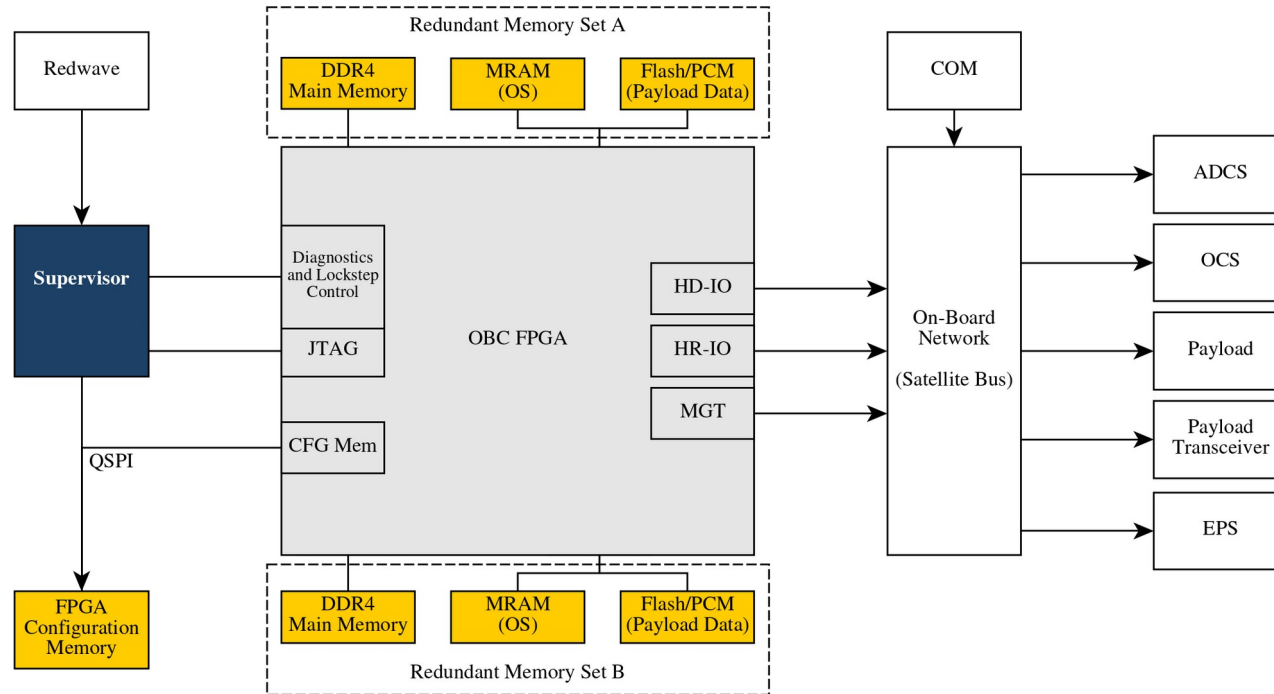
Testing Avionics Architectures running Software-implemented Fault Tolerance

By Christian M. Fuchs

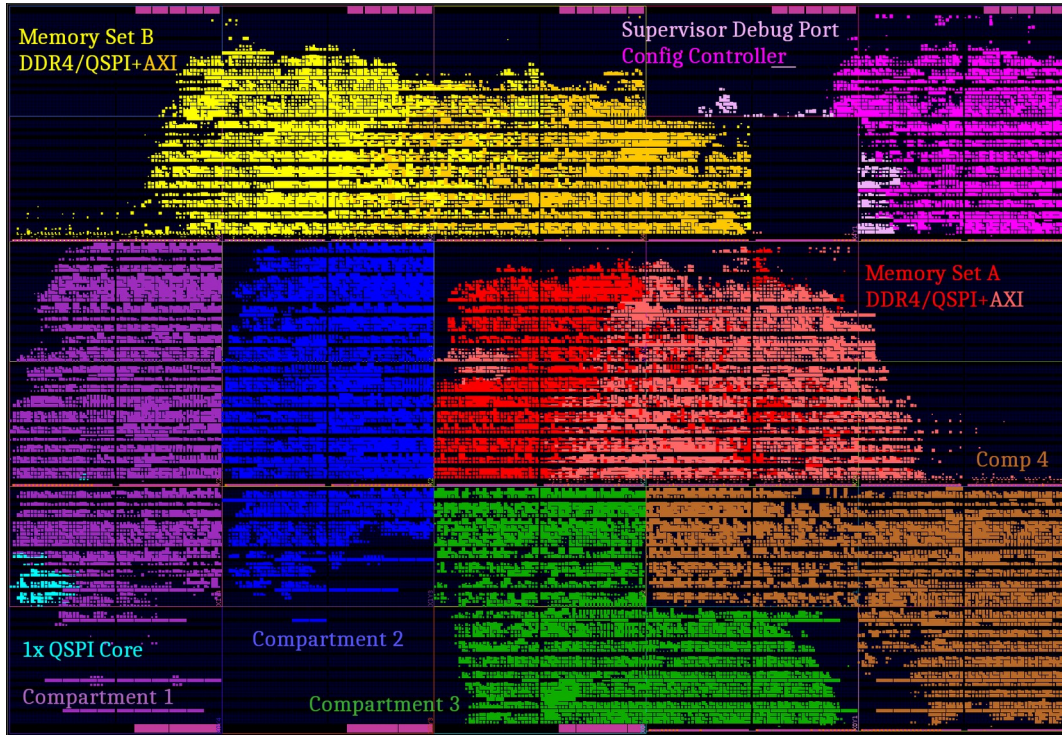
Background

- Developed a fault-tolerant computer architecture suitable for small satellites, which shall:
 - Run standard “rich” applications – e.g. Linux, ...
 - Be suitable for ~2U CubeSats and heavier satellites
 - Be built with just COTS components and standard IP
 - Only use “ordinary” tech, no RHBM/space-grade parts
 - Handle semiconductor degradation and age gracefully

Platform Architecture



MPSoC Implementation

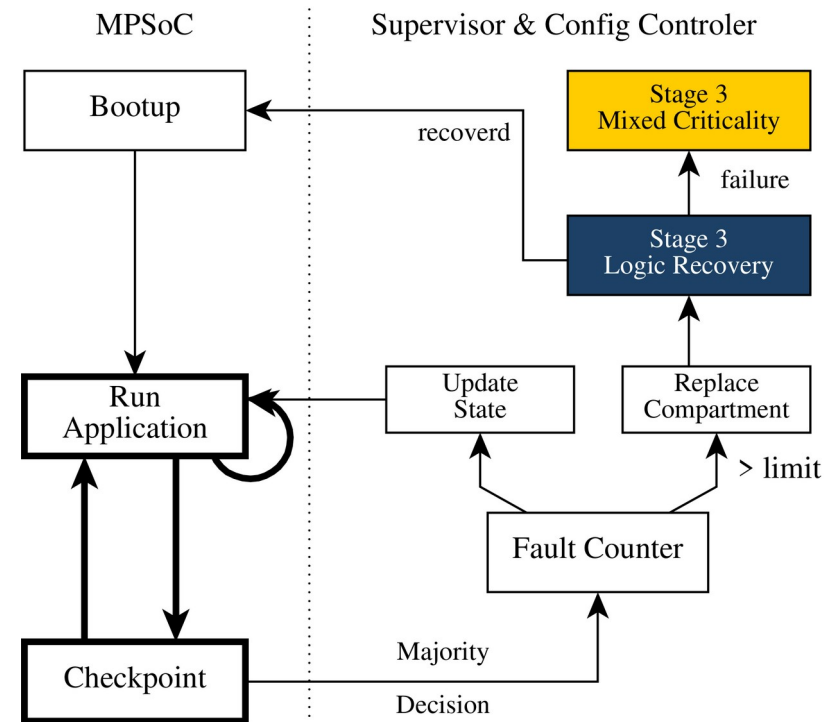


Developed designs for

- Kintex Ultrascale KU060
- Kintex Ultrascale+
 - KU11P, KU3P
 - KU5P on KCU116
- Virtex Ultrascale+
 - VU9P on VCU118

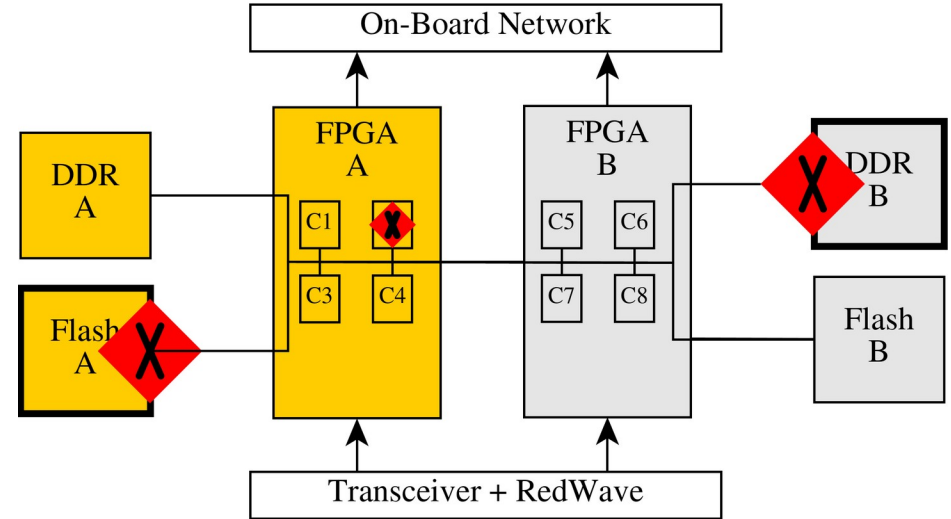
Multistage Fault-Tolerance

- Stage 1 – Coarse-Grain Thread-Level Lockstep
 - Acts as FDIR trigger and
 - Assure fault-coverage immediately
- Stage 2 – Autonomous Compartment Recovery
 - Reconfigure FPGA to recover parts of MPSoC
 - Dynamic Application Migration for non-stop operation
- Stage 3 – Fault-Adaptive Software Mapping
 - Adapt to shrinking resource pool as permanent faults accumulate
 - Exploit Mixed Criticality to achieve Graceful Aging



Multistage Fault-Tolerance

- Stage 1 – Coarse-Grain Thread-Level Lockstep
 - Acts as FDIR trigger and
 - Assure fault-coverage immediately
- Stage 2 – Autonomous Compartment Recovery
 - Reconfigure FPGA to recover parts of MPSoC
 - Dynamic Application Migration for non-stop operation
- Stage 3 – Fault-Adaptive Software Mapping
 - Adapt to shrinking resource pool as permanent faults accumulate
 - Exploit Mixed Criticality to achieve Graceful Aging



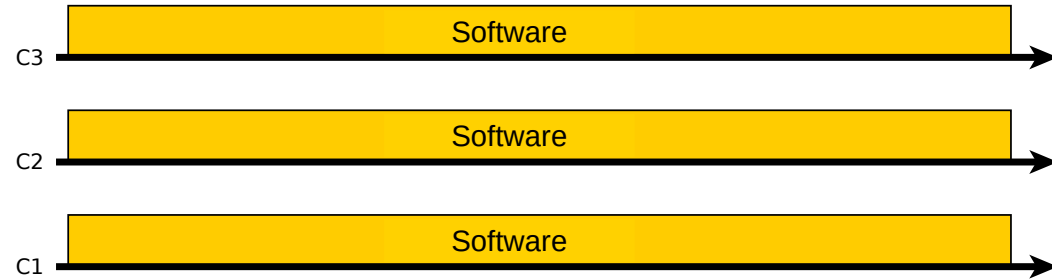
Coarse Grain Lockstep

- Consider how software runs on a computer
- Imagine a processor executing software
- On an MPSoC, multiple instances of an application can be run in parallel and run in lockstep



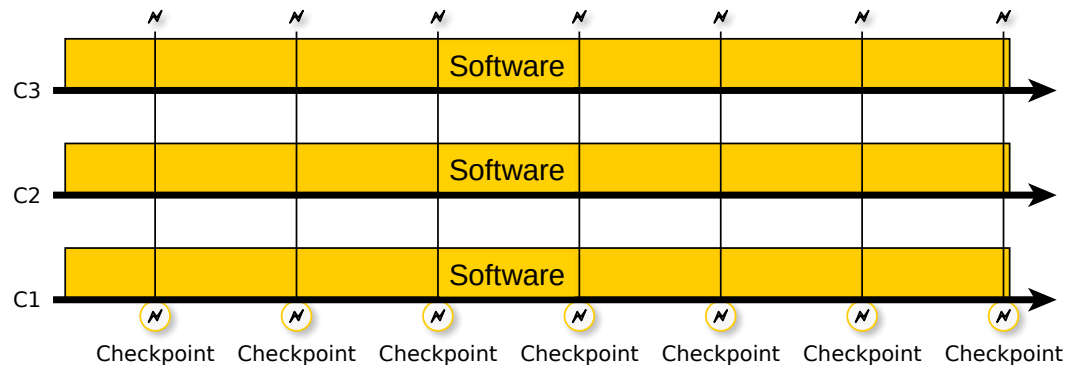
Coarse Grain Lockstep

- On an MPSoC, multiple instances of an application can be run in parallel and run in lockstep
- Assuming the Architecture allows that.
 - Lockstep in Hardware
 - Has scalability issues
 - Lockstep in Software
 - Can be done with any commercial IP
 - Does not suffer from routability issues
 - Less residual “critical” logic
 - Improved Diagnostics



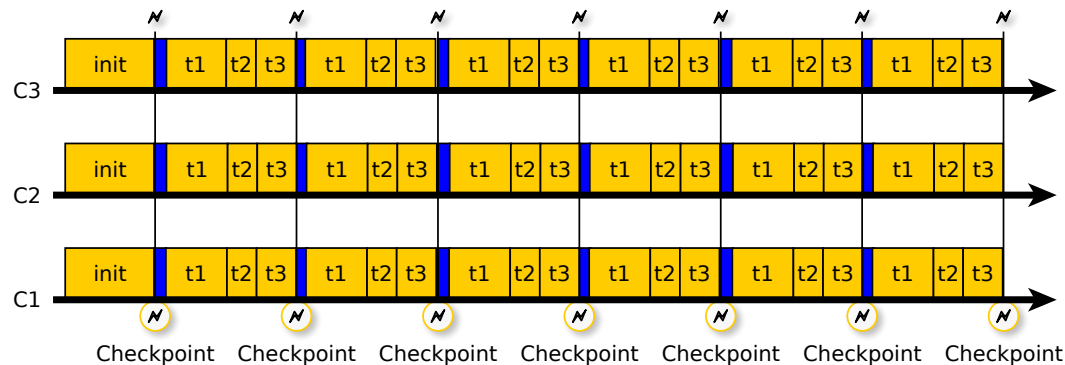
Coarse Grain Lockstep

- Introduce Checkpoints during regular operation
- Compare state of Software
- Achieve Majority Decision
- Checkpoints are triggered externally (interrupt) or internally (time)
- Settings Run-time configurable



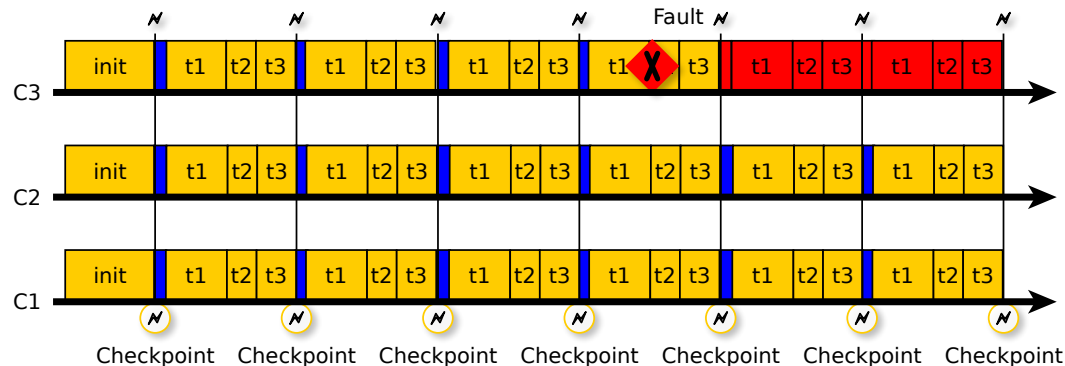
Coarse Grain Lockstep

- Introduce Checkpoints during regular operation
- Compare state of Software
- Achieve Majority Decision
- Checkpoints are triggered externally (interrupt) or internally (time)
- Settings Run-time configurable



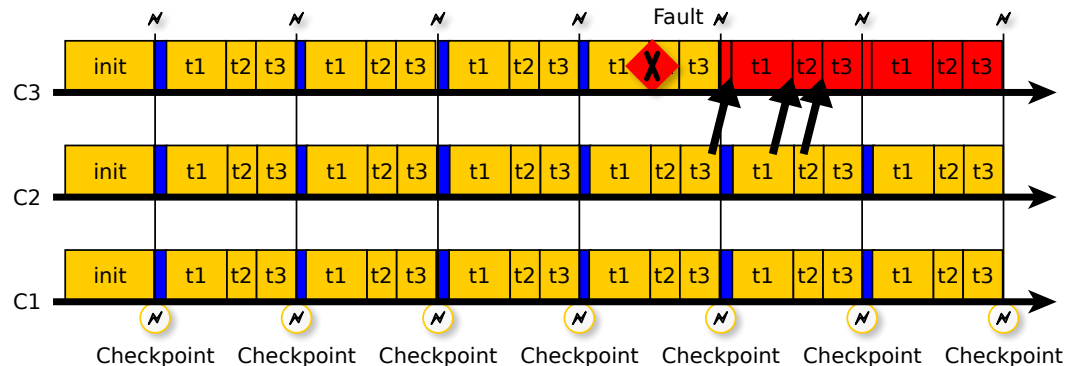
Coarse Grain Lockstep

- Faults will change the system state, cause crashes, or otherwise alter how a processor executes software.
- This setup enables Fault-Detection
- To recovery from soft-errors, copy state from another compartment
- Limitation:
 - Not all faults can be recovered from using state updates
 - Reboots would cause delays
 - Spares requires to handle permanent faults



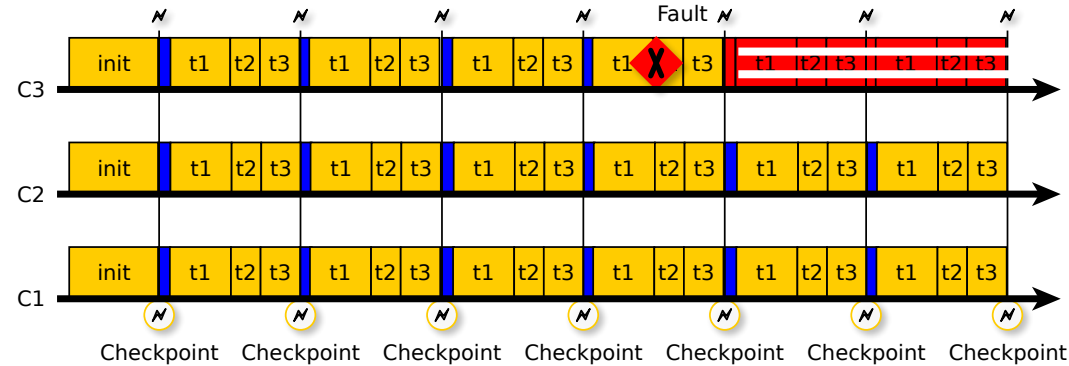
Coarse Grain Lockstep

- Faults will change the system state, cause crashes, or otherwise alter how a processor executes software.
- This setup enables Fault-Detection
- To recovery from soft-errors, copy state from another compartment
- Limitation:
 - Not all faults can be recovered from using state updates
 - Reboots would cause delays
 - Spares requires to handle permanent faults



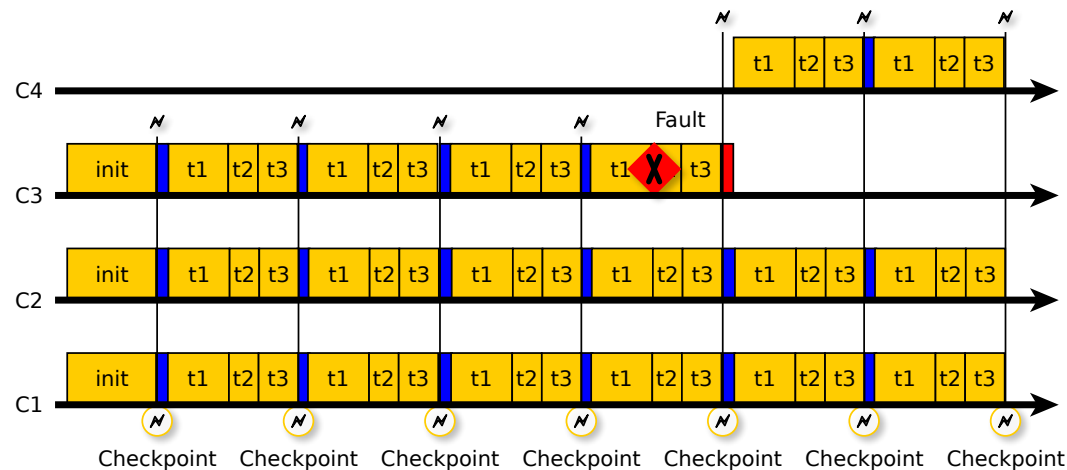
Coarse Grain Lockstep

- Permanent faults in a compartment can be handled through software replication
- To replace a failed compartment, provision a replacement
- Migrate application copies to replacement
- Apply same mechanics for state update as before
- Remove/reboot/repair failed part



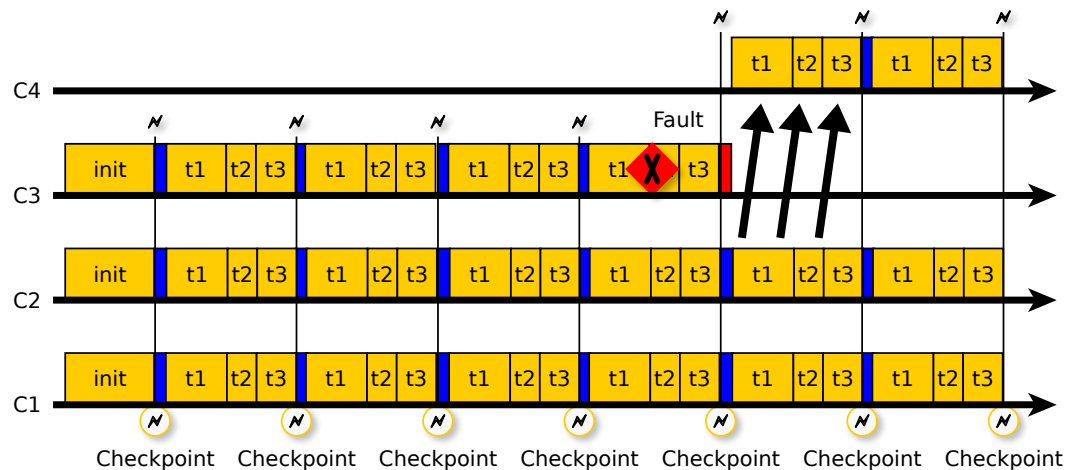
Coarse Grain Lockstep

- Permanent faults in a compartment can be handled through software replication
- To replace a failed compartment, provision a replacement
- Migrate application copies to replacement
- Apply same mechanics for state update as before
- Remove/reboot/repair failed part

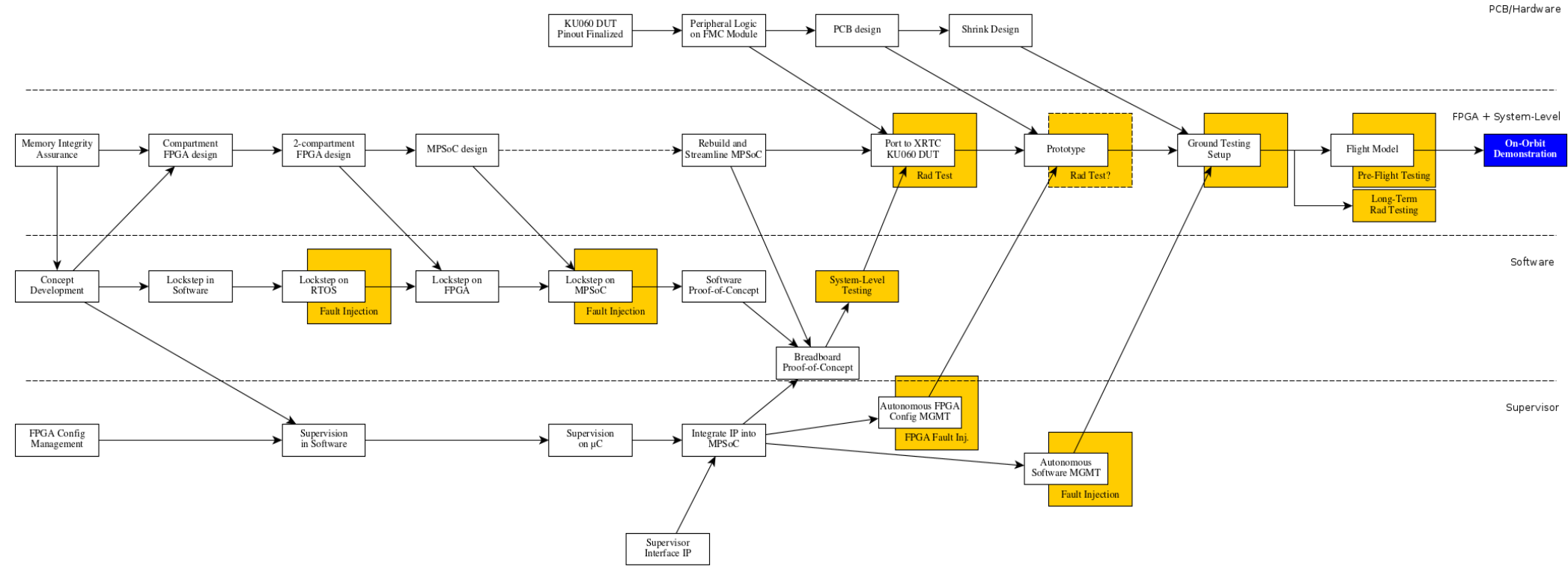


Coarse Grain Lockstep

- Permanent faults in a compartment can be handled through software replication
- To replace a failed compartment, provision a replacement
- Migrate application copies to replacement
- Apply same mechanics for state update as before
- Remove/reboot/repair failed part

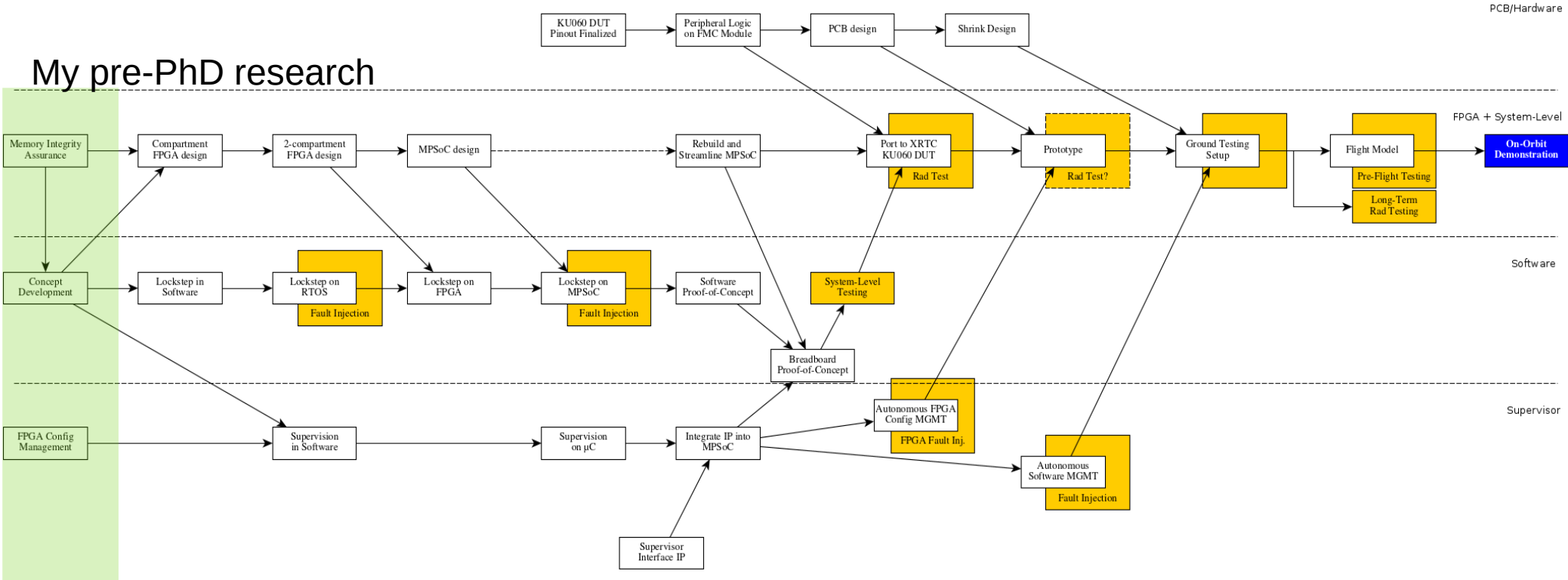


Development Roadmap

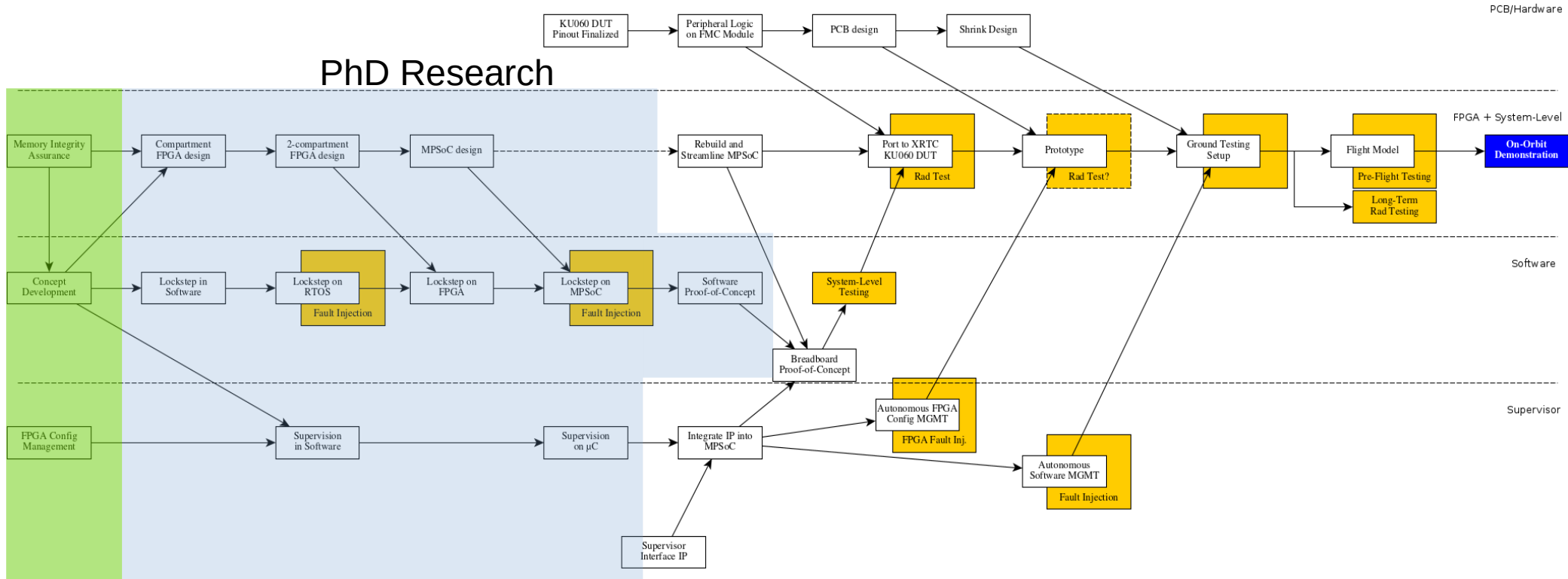


Development Roadmap

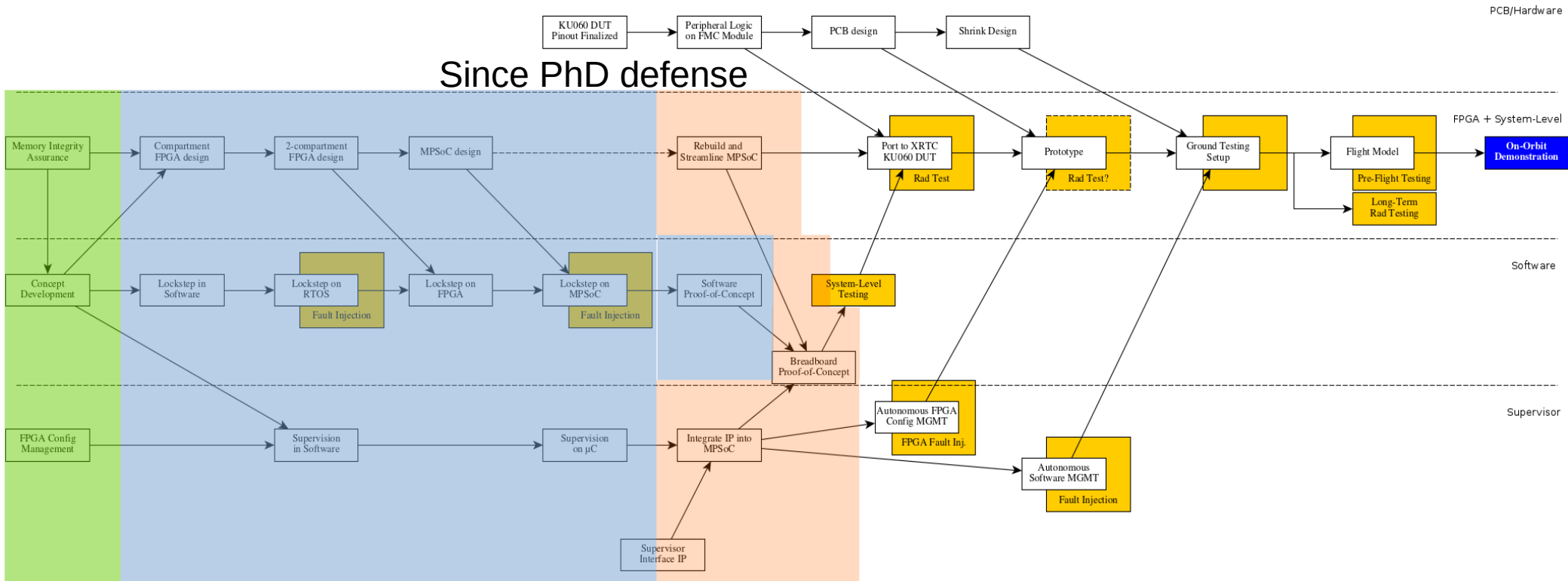
My pre-PhD research



Development Roadmap



Development Roadmap



How to validate Software Fault Tolerance

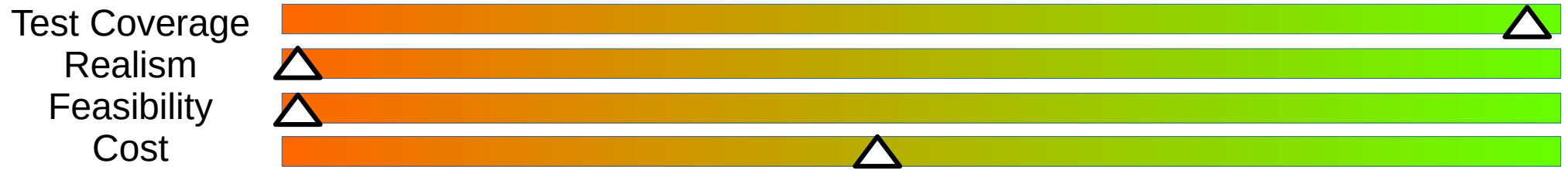
- Formal Verification
- System-Level Testing
- System Simulation at different Levels

Disclaimer: as seen from our (subjective!) perspective



Formal Verification & Proofs

- Assert mathematical or formal properties
- Caveats:
 - Extreme amount of test preparation necessary
 - Obtained data can be difficult to interpret
 - Technically only feasible for
 - Small IP cores
 - Abstract algorithms
 - Simple application software



System-Level Fault Injection

- Against real Hardware-Prototype
 - Requires a suitable hardware prototype ...
 - Unfortunately, we can not import one from the future.
- Caveats:
 - Potentially destructive
 - High cost
 - Not practically feasible at this stage of development



System-Level Fault Injection

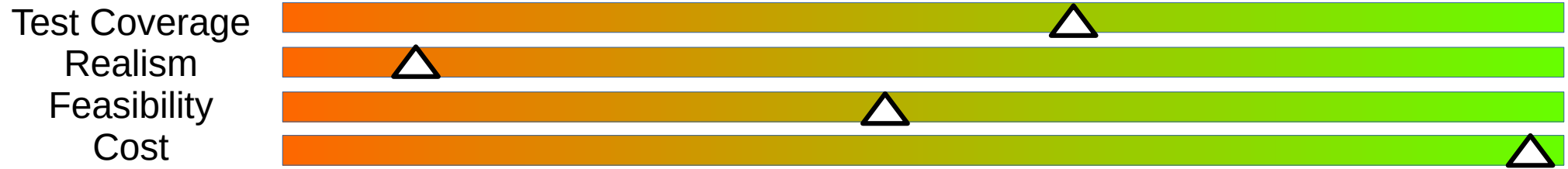
- Port Architecture to FPGA
 - Great, what we need to test exists as MPSoC design.
- Caveats:
 - The FPGA part is just one component among several
 - Within a development board, MPSoC topology differs
 - Impossible to achieve reasonable level of test coverage, as we run a full OS



Software Fault Injection

- Inject faults programmatically e.g., add them to the code
- Run target application as application
- Caveats:
 - Requires modification to the application
 - Not viable for close to hardware or kernel-level functionality
 - Unless executed directly on hardware (potentially destructive)

```
ProgressiveLinearLeastSquaresFit(...) {  
    [...]  
    for (y = 0; y < FRAME_DIM_Y; y++) {  
        for (x = 0; x < FRAME_DIM_X; x++) {  
            If ( FAULT || saturationFrame[y][(x >> 5)] == 0) {  
                frameMinusOffset = lastframe[y][x] - offsetCosmicFrame[y][x];  
                sumXYFrame[y][x] = N * frameMinusOffset + 0;  
                sumYFrame[y][x] = frameMinusOffset + sumYFrame[y][x];  
            } else if ( [...] ) {  
                frameMinusOffset = frame[y][x] - offsetCosmicFrame[y][x];  
                sumXYFrame[y][x] = N * frameMinusOffset + sumXYFrame[y][x];  
                sumYFrame[y][x] = frameMinusOffset + sumYFrame[y][x];  
            } else {  
                [...]  
            }  
        }  
    }  
    [...]  
    return &result;  
}
```



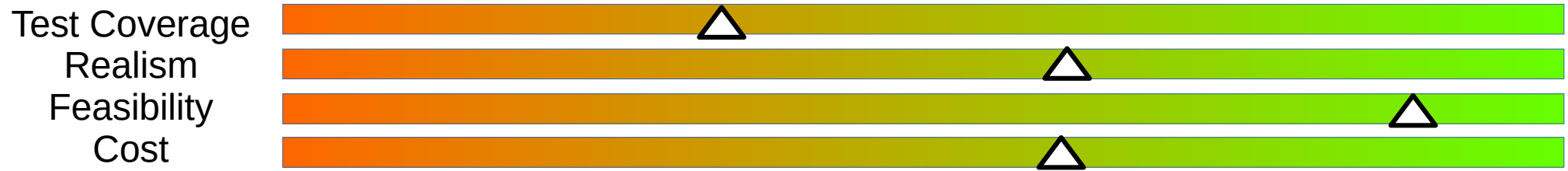
Debugger-Based Fault Injection in Userspace

- Attach debugger to application to inject faults
- Test setup runs as userspace application on simulation host
- Caveats:
 - Only soft-errors can be represented without major obstacles
 - Debugger interface is bottleneck
 - Test environment is different than on real system



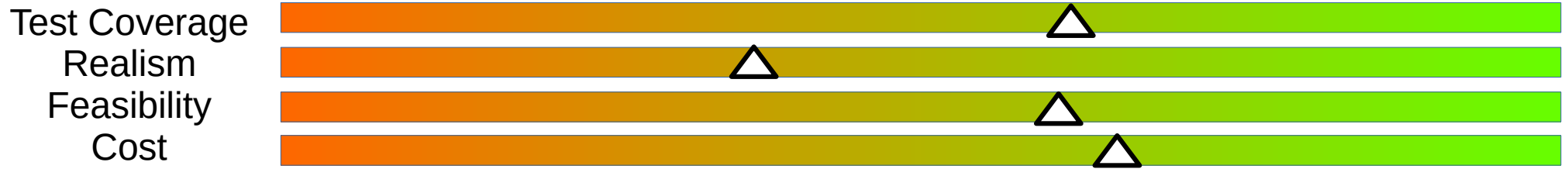
Debugger-Based Fault Injection into Metal

- Attach debugger to software running directly on hardware
- Test setup runs as full OS/RTOS/firmware/etc ...
- Caveats:
 - Only soft-errors can be represented without major obstacles
 - Debugger interface is bottleneck
 - Potentially destructive and long dead-times



Fault Injection using Binary Introspection

- Inject code at runtime to simulate complex fault behavior to improve result quality
- Test application software in userspace or on metal
- Caveats:
 - Injected code is non-trivial (akin to writing exploit code)
 - Test environment is still nowhere near a real system

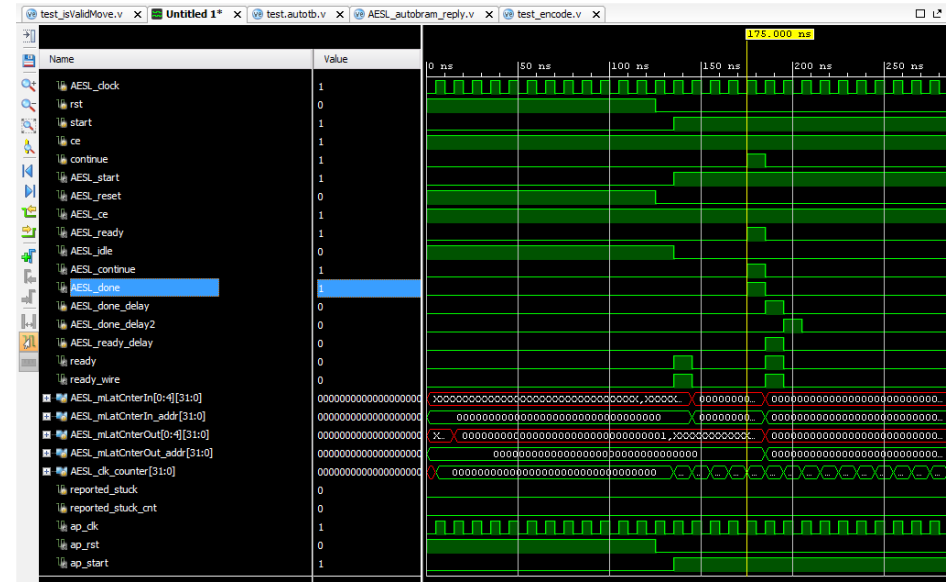


System Simulation

- Use behavioral models to simulate, emulate, or virtualize a system and run “real” lockstep
- Several flavors with different capabilities:
 - RTL/Netlist Simulation
 - SystemC Simulation
 - Virtualization and virtual machine introspection
 - ISA level fault injection

Netlist Simulation

- Simulate architecture at netlist/gate-level
- Caveats:
 - Computationally extremely expensive
 - Impossible to run real OS or simulate full MPSoC



Test Coverage



Realism



Feasibility

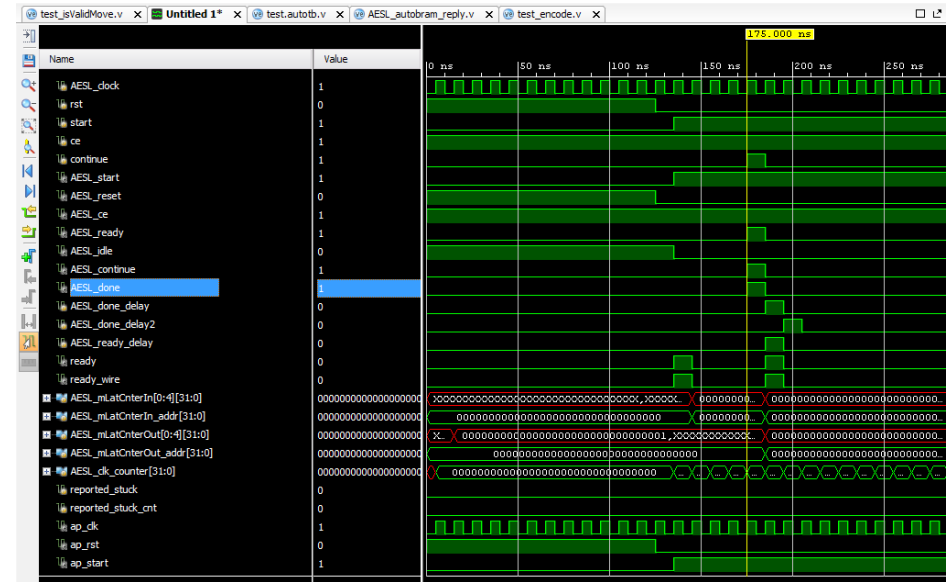


Cost



RTL Simulation

- Simulate architecture at register/logic-level
- Caveats:
 - Computationally very expensive
 - Practically impossible to run real OS or simulate full MPSoC



Test Coverage



Realism



Feasibility

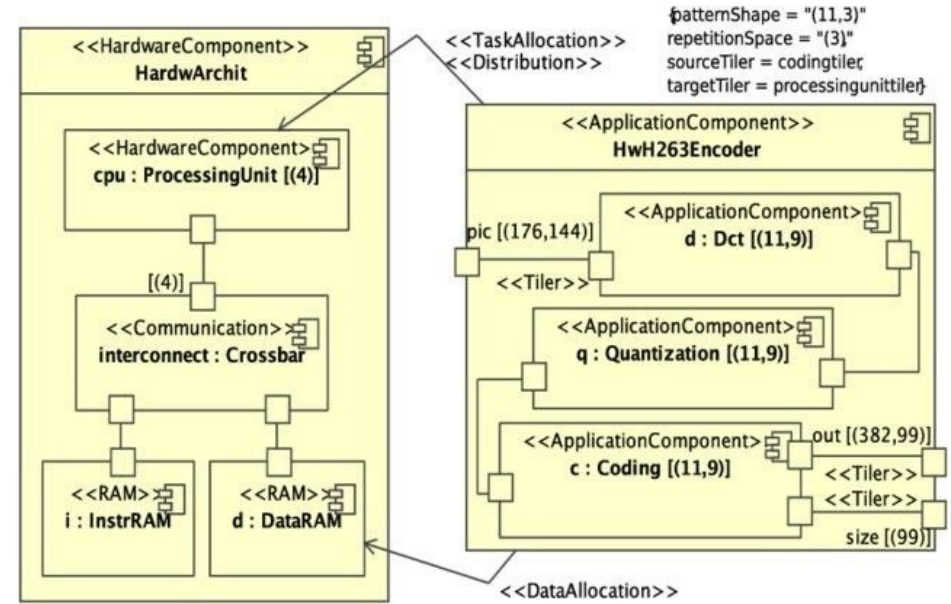


Cost



SystemC Fault Simulation

- Compile system-simulator from of set of behavioral models
 - Can properly simulate multi-core systems
 - Much more accurate/realistic than SW-FI
 - Drastically faster than RTL/netlist simulation
 - Caveats:
 - System model composition is labor intensive and difficult to debug
 - Tool & model availability
- Requires high-level system modeling language to really take advantage of (e.g. ArchC)



Atitallah et. al.: "From high level MPSoC description to SystemC code generation", 2007

Test Coverage

Realism

Feasibility

Cost



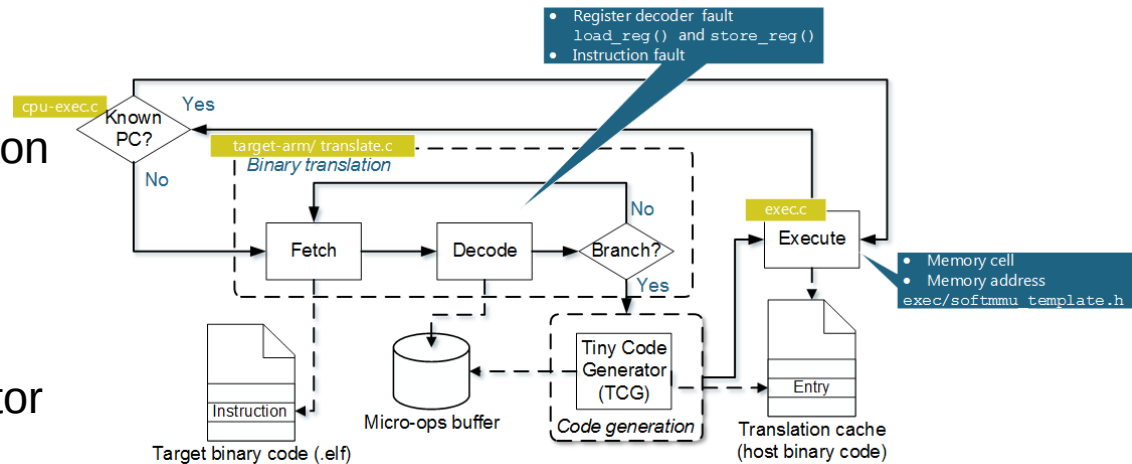
Debugger-Based Fault Injection into a Virtual Machine

- Attach debugger to Virtual Machine to inject faults, ...
- ... or utilize Virtual Machine Introspection
- Caveats:
 - Only soft-errors can be represented without major obstacles
 - Debugger interface is bottleneck (even more)
 - Quality of results depends on quality of emulator/VMM/Hypervisor
 - FI into multi-core systems may break timing and is non-trivial



ISA-level fault injection

- Emulate System Functionality and Inject faults directly
 - Faster than RTL/SystemC Simulation
 - Open source tools available
- Limitations
 - Accuracy depends on used Emulator
 - Multiprocessor Simulation Limited
 - Bias towards crashes instead of “interesting” results



Metric	Segment	Marker Position (approx. %)
Test Coverage	Green	85
Realism	Orange	65
Feasibility	Green	95
Cost	Green	80

ISA-level Fault Simulation

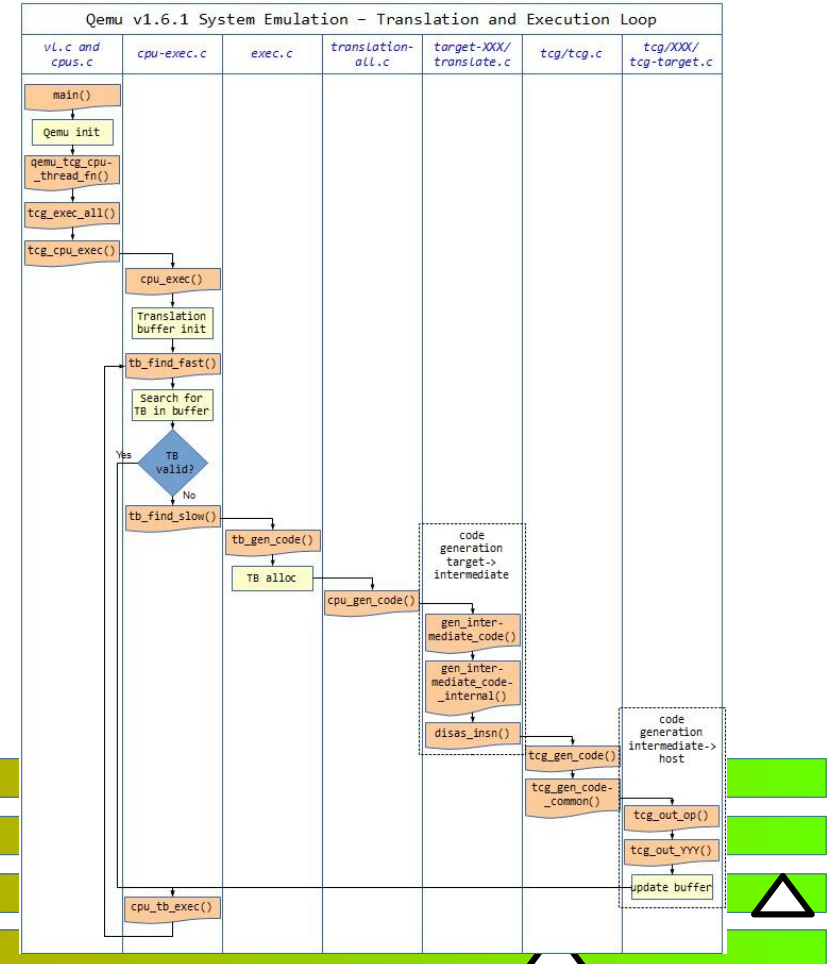
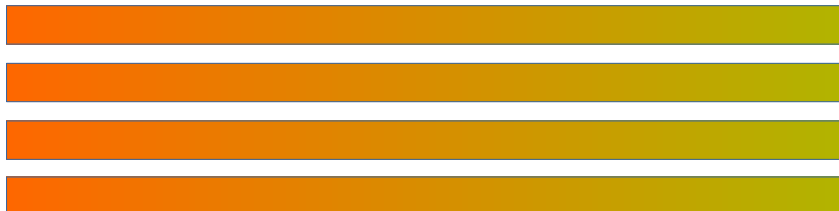
- Emulate System Functionality and Inject faults directly
 - Faster than RTL/SystemC Simulation
 - Open source tools available
- Limitations
 - Accuracy depends on used Emulator
 - Multiprocessor Simulation Limited
 - Bias towards crashes instead of “interesting” results

Test Coverage

Realism

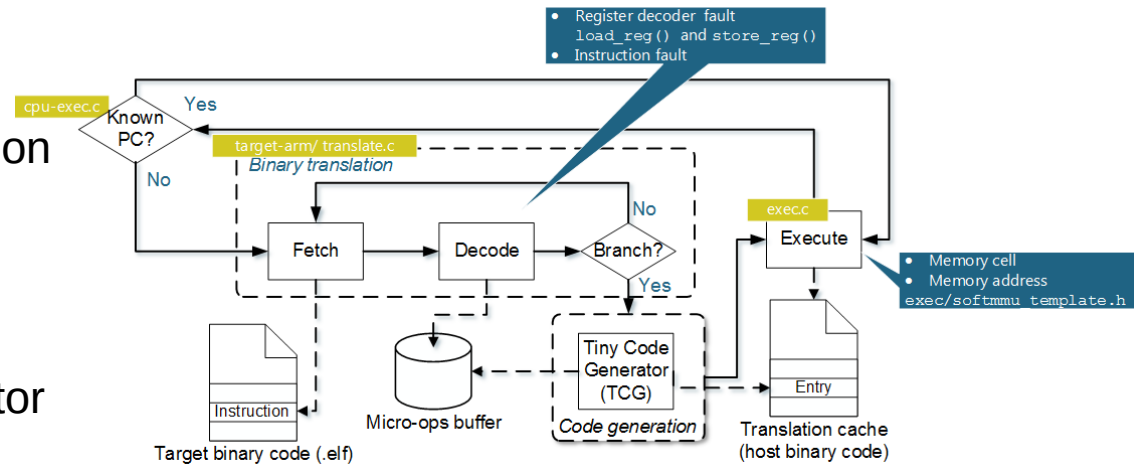
Feasibility

Cost



ISA-level Fault Simulation

- Emulate System Functionality and Inject faults directly
 - Faster than RTL/SystemC Simulation
 - Open source tools available
- Limitations
 - Accuracy depends on used Emulator
 - Multiprocessor Simulation Limited
 - Bias towards crashes instead of “interesting” results



Holler et. al.: "Advances in Software-Based Fault Tolerance for Resilient Embedded Systems", 2017

Metric	Segment	Marker Position (approx. %)
Test Coverage	Olive Green	85
Realism	Boundary (Olive/Light Green)	65
Feasibility	Light Green	95
Cost	Boundary (Olive/Light Green)	80

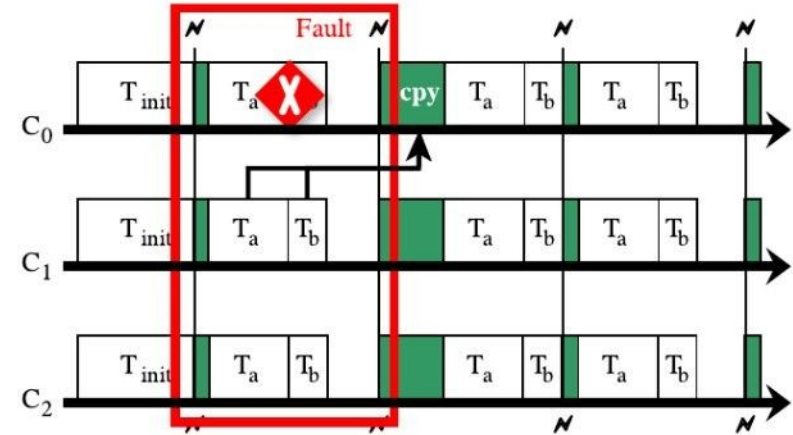
Our Approach – Use 3 Methods

- Software fault Injection
 - To test for functionality
 - Results not worth publishing due to simple testbench and low realism
- ISA-level fault injection
 - To test fault detection capability, voter stability, and check code overhead
 - into RTOS Implementation of Lockstep
- SystemC fault injection
 - into MPSoC-Model running Lockstep
 - To test fault recovery and assure that ISA-level results are realistic
 - Performance with/without memory isolation between compartments

Currently unpublished, looking for decent place to publish

NOTE:

- We are not interested in the behavior of the payload application, but rather in the capability of the lockstep to detect, isolate, and recovery from faults that have occurred and cause the payload application's state to become corrupted.

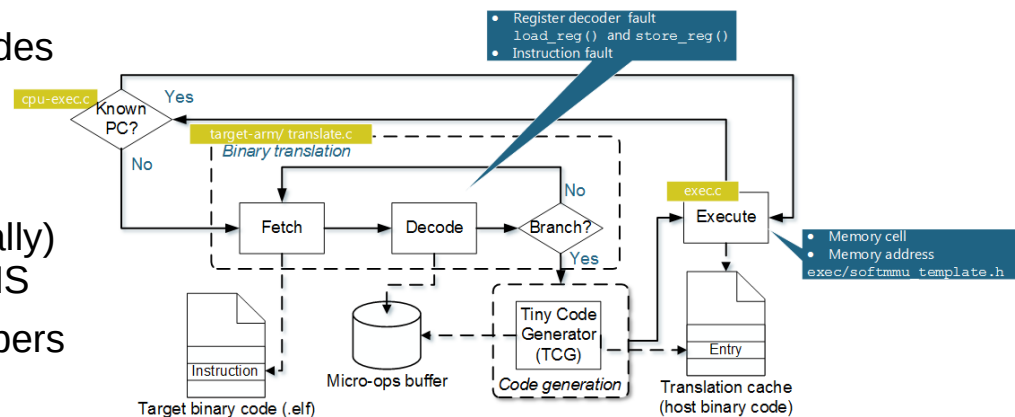


ISA Level Fault Simulation

- Fault Simulation using FIES by Holler et al.
 - Based on industry standard tool QEMU
 - Very academic tool, I rewrote most of it
- Source code available at:
<https://github.com/dependableDOTspace/FIESer>
- Added support for necessary sub-ISAs and fault modes

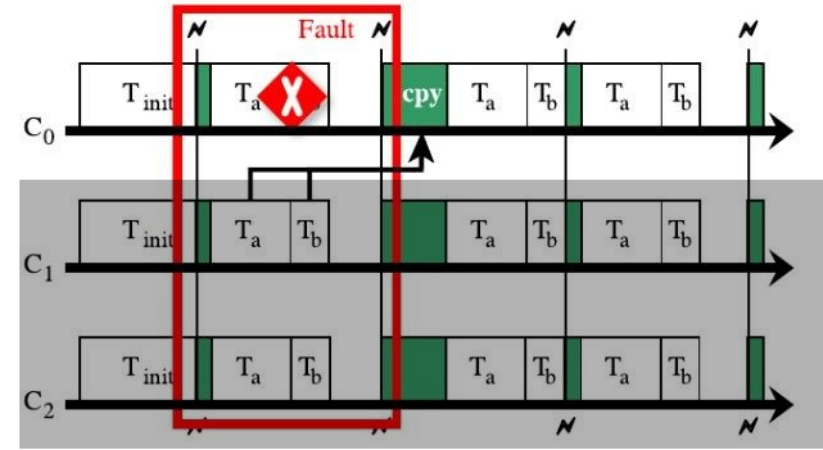


- Test Setup
 - 1 Victim Core running JWST/MIRI-like payload (initially) ESA's NextGen Space DSP Benchmark GCC/RTMEMS
 - Majority Voted Lockstep Setup with 4 lockstep members
 - Compared System Behavior to fault-free simulation



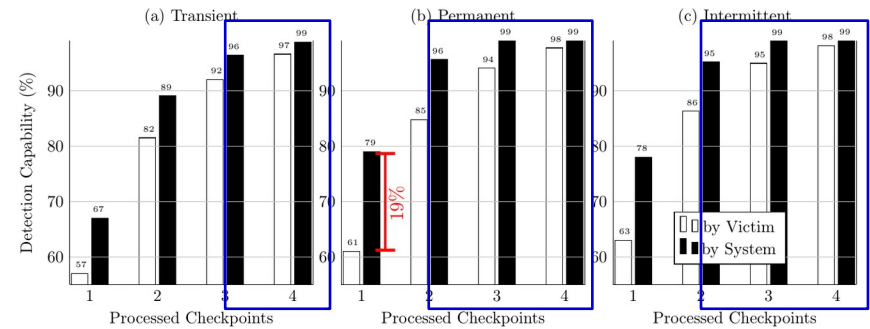
ISA Level Fault Simulation

- Fault Simulation using FIES by Holler et al.
 - Based on industry standard tool QEMU
 - Very academic tool, I rewrote most of itSource code available at:
<https://github.com/dependableDOTspace/FIESer>
 - Added support for necessary sub-ISAs and fault modes
- Test Setup
 - 1 Victim Core running JWST/MIRI-like payload (initially) ESA's NextGen Space DSP Benchmark GCC/RTEMS
 - Majority Voted Lockstep Setup with 4 lockstep members
 - Compared System Behavior to fault-free simulation



Fault Detection Capability – ISA

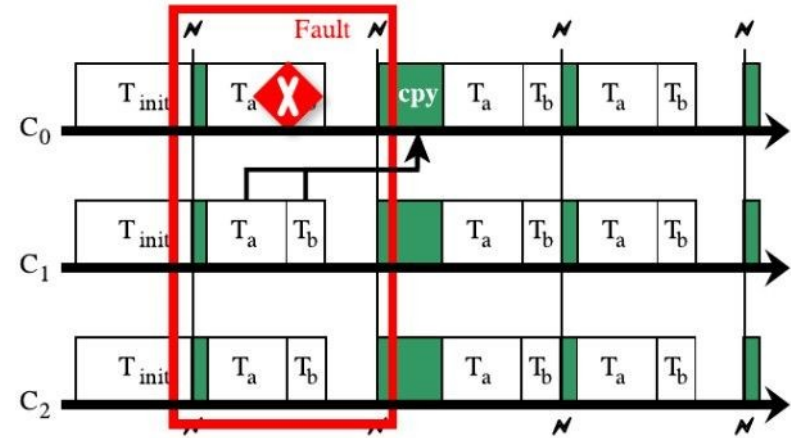
- Key Questions of Interest to us:
 - Voter Stability?
 - Sufficiently high to avoid oscillation effects
 - Time to detect a fault?
 - After ~3 Checkpoints with near certainty



Impact	Fault	Detectable by		Recovery through	Observed Effect per Fault Type		
	Detectable	victim tile	other tiles		Transient	Permanent	Intermittent
Corrupted Thread State	yes	yes	yes	state-update	49%	44%	53%
Thread Crash	yes	yes	no	state-update	8%	17%	10%
Lockstep Failure	yes	no	yes	reboot	1%	2%	1%
OS Crash	yes	no	yes	reboot	10%	18%	15%
No Effect (SDC/Masked)	no	no	no	reboot	32%	19%	21%

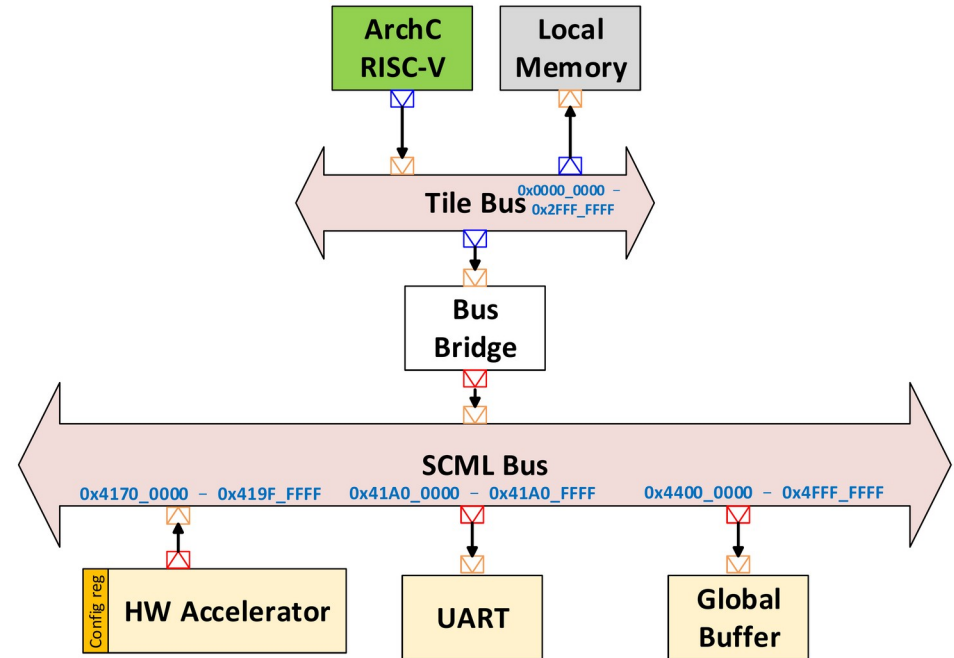
SystemC Fault Simulation

- True multi-core setup modeled using ArchC
 - Collaboration with NTHU, Taiwan
 - Simplified MPSoC architecture to avoid runaway test time, but still run lockstep and payload software
 - Topological isolation between compartments
- Test Setup
 - 3-core system running ESA's NextGen Space DSP Benchmark (GCC/baremetal)
 - Core recovery through lockstep, but no core replacement
 - Fault-propagation time up to 1min
- Test-space reduction using ACE/AVF



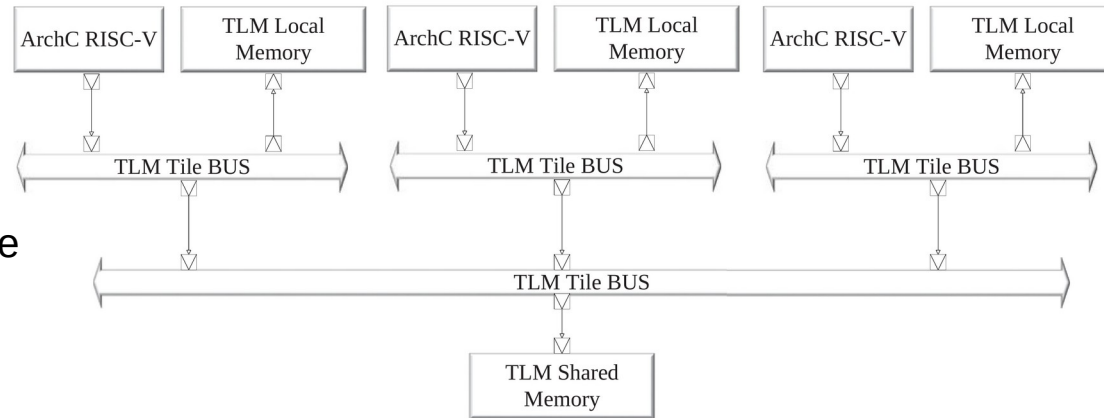
SystemC Fault Simulation

- True multi-core setup modeled using ArchC
 - Collaboration with NTHU, Taiwan
 - Simplified MPSoC architecture to avoid runaway test time, but still run lockstep and payload software
 - Topological isolation between compartments
- Test Setup
 - 3-core system running ESA's NextGen Space DSP Benchmark (GCC/baremetal)
 - Core recovery through lockstep, but no core replacement
 - Fault-propagation time up to 1min
- Test-space reduction using ACE/AVF



SystemC Fault Simulation

- True multi-core setup modeled using ArchC
 - Collaboration with NTHU, Taiwan
 - Simplified MPSoC architecture to avoid runaway test time, but still run lockstep and payload software
 - Topological isolation between compartments
- Test Setup
 - 3-core system running ESA's NextGen Space DSP Benchmark (GCC/baremetal)
 - Core recovery through lockstep, but no core replacement
 - Fault-propagation time up to 1min
- Test-space reduction using ACE/AVF



Fault Detection Capacity ISA vs SystemC

- Lockstep Implementation Differences:
 - RTOS vs Baremetal
 - Pthreads vs non-threaded
- Some deviation to be expected
- Do the results roughly match?
- Is the lockstep's performance still comparable?

Result	Effect by Injected Fault Type			
	FIES	ArchC	FIES	
	Transient	Transient	Permanent	Intermittent
Corrupted State	49%	32%	44%	53%
Thread Crash	8%	-	17%	10%
Lockstep Failure	1%	1%	2%	1%
Crash/Hangup	10%	14%	18%	15%
No Effect/SDC	32%	54%	19%	21%

Fault Detection Capacity ISA vs SystemC

- Lockstep Implementation Differences:
 - RTOS vs Baremetal
 - Pthreads vs non-threaded
- Some deviation to be expected
- Do the results roughly match?
- Is the lockstep's performance still comparable?
- Runtime:
 - ISA-level runtime: 45s
 - Original SystemC runtime: 7h
 - Optimized SystemC runtime: 10s

	Ref.	FIES @ 54% SDC	ArchC	Δ
Corrupted State	49%	38.22%	31.72%	-6.5%
Thread Crash	8%	6.24%	0%	-6.24%
Lockstep Failure	1%	1%	1%	0%
Crash/Hangup	10%	7.8%	14.54%	+7.66%
			Δ Total	5.08%

Lockstep Recovery

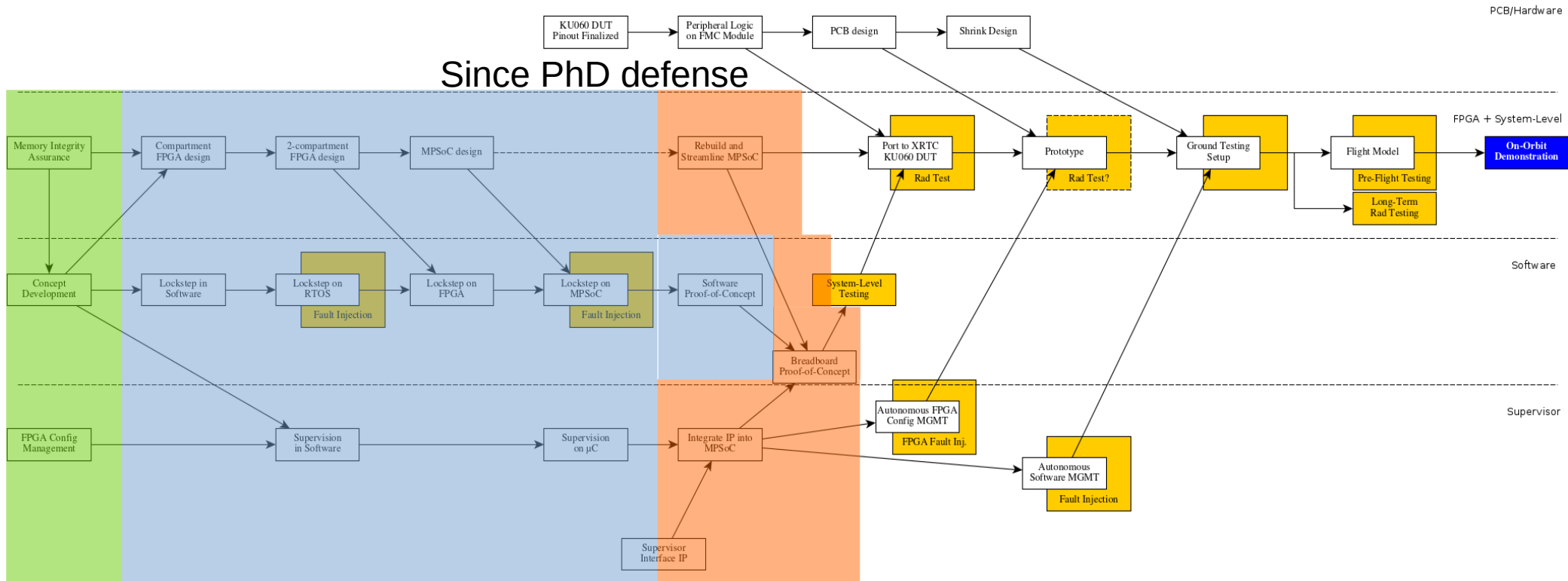
- Once the lockstep detects that a fault occurred:
 - Can we recover right away through a state update?
 - How likely is it direct supervisor intervention is required?
 - Does a compartment need to be replaced?

Immediate Recovery	Lockstep Timeout	Reboot Required
22004 46%	10915 23%	14607 31%

Deductions and Lessons Learned

- Fault injection results between ISA level and SystemC match closely
 - Differences in application structure have little to no impact on lock step behavior
 - Difference in fault injection results about 5%
 - SystemC results more trustworthy as verified against RTL simulation
- Permanent faults are easier to detect than transients
 - But severity shifts, more faults that cause state corruption now cause crashes
- Threading overall reduces likelihood that an entire OS will crash
- During SystemC model development, we “forgot” to implement memory isolation between compartments
 - Noticed due to “strange” failures in first fault injection runs
 - System topology has a noticeable effect on lockstep

Development Roadmap

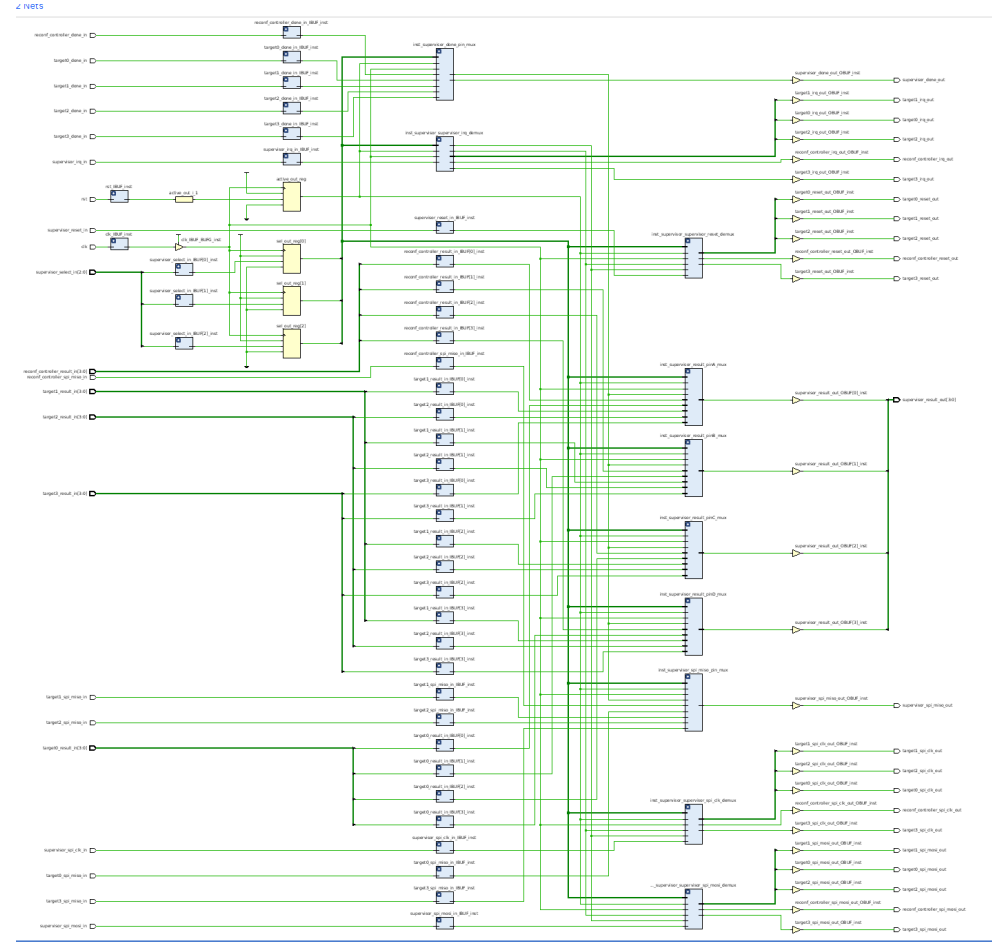


What Happened Since Q4/2019

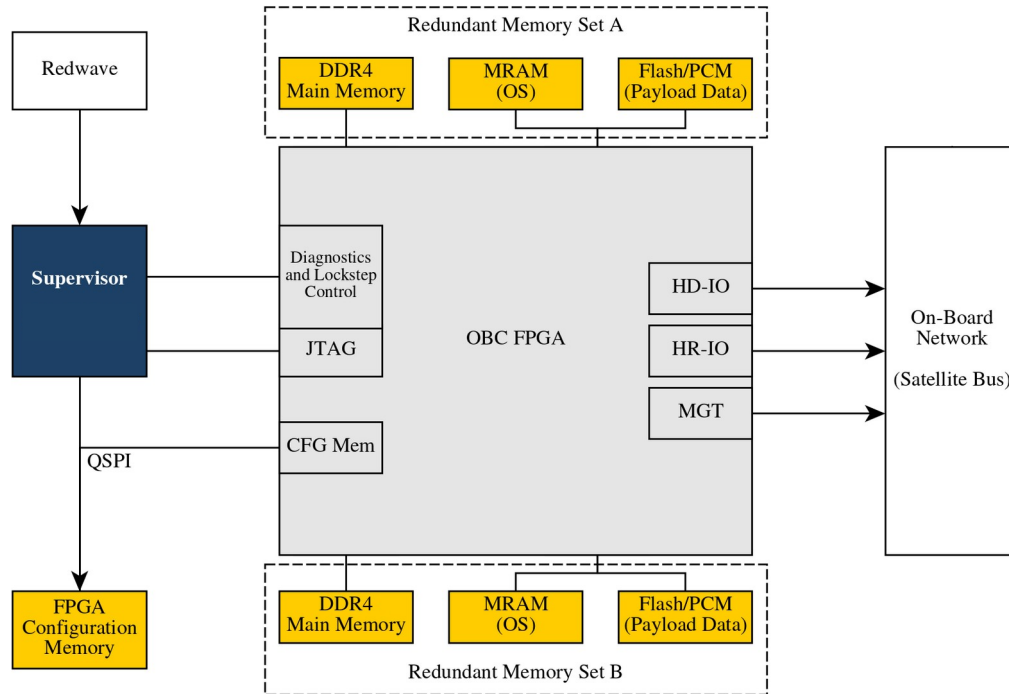
- Complete redesign of MPSoC to make it ready to move to KU060 DUT Card
- Implementation of the supervisor interface IP
 - SPI + Result Register + Rest and Interrupt Signals
 - From supervisor to compartments and PR controller
- Integration of the supervisor interface IP into MPSoC
 - Considerable time spent on testing and debugging testbenches
 - Even basic IO functionality of Xilinx Eval Kit Designs is definitely not bug-free...
- Functional breadboard set up
 - “The system runs!”
 - Getting ready for System-level Testing
 - Now FPGA fault injection begins to make sense

Supervisor Control Interface

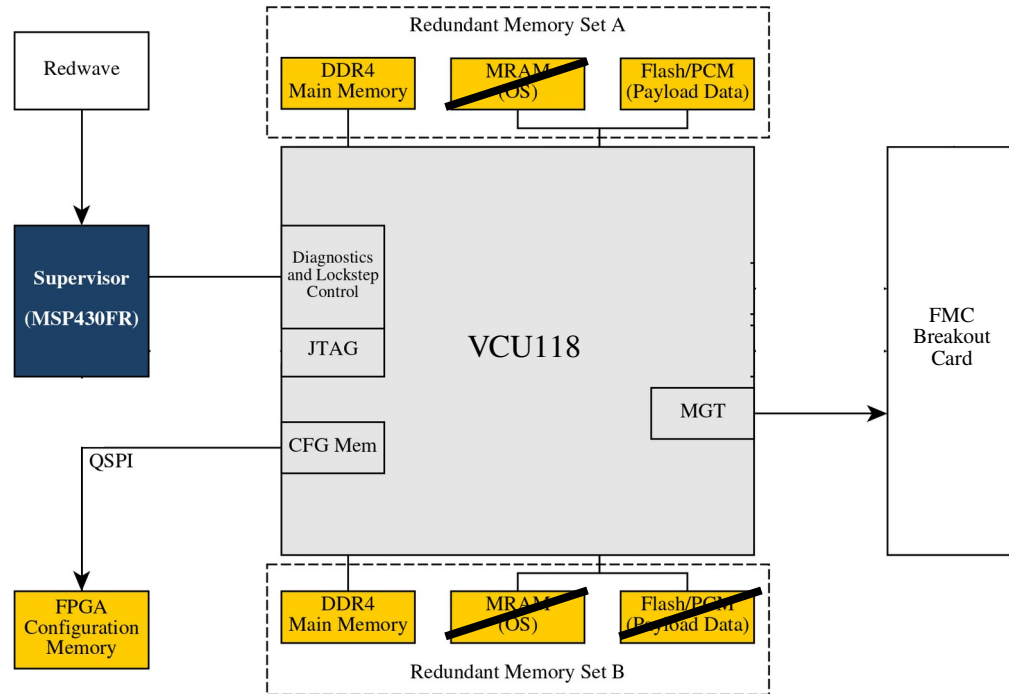
- Low-complexity Supervisor Interface:
 - Trigger interrupts and reset in compartments
 - Access each compartment address space via AXI bridge
 - Control over reconfiguration controller
 - Get notify about system events
 - Monitor voting results
- Essentially glorified MUX/DeMUX array
 - SPI, Interrupt, Reset, Voting Results
 - Reset and Interrupt multi-target support



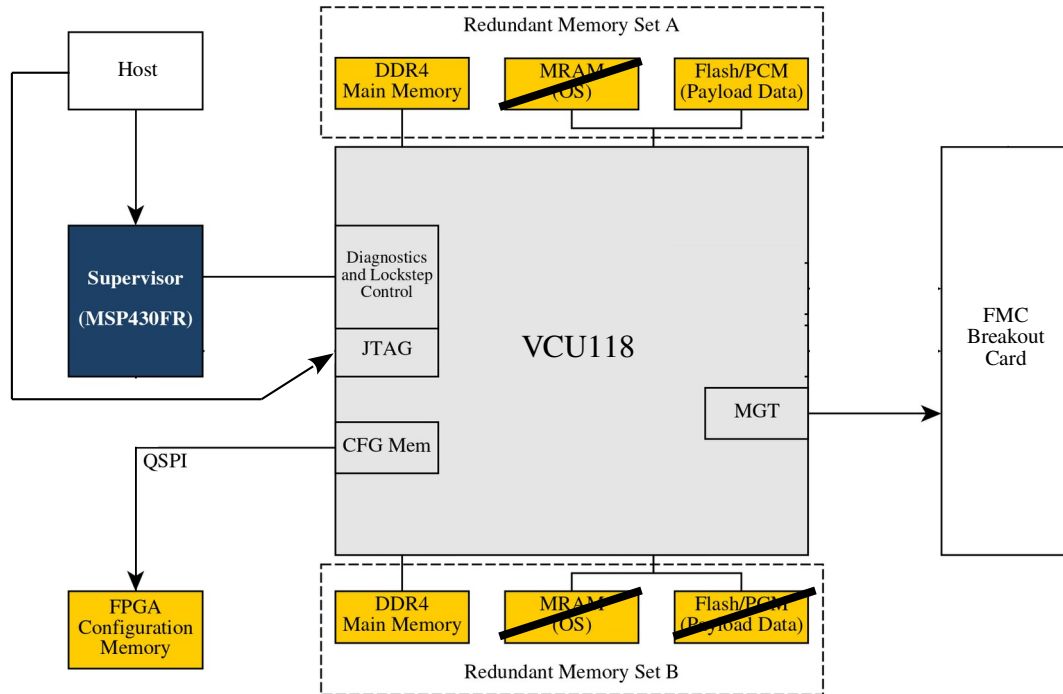
Platform Architecture



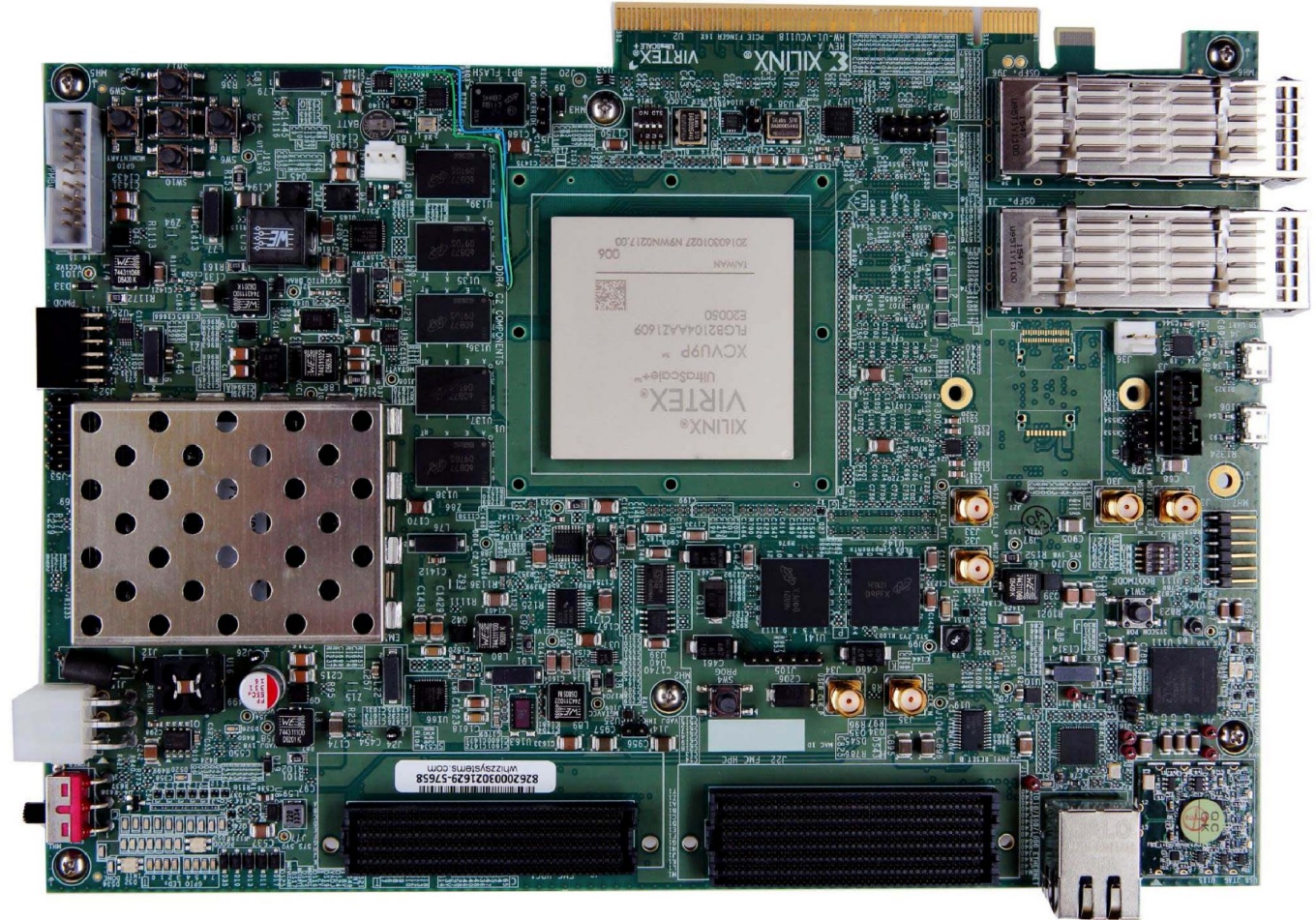
Current Proof-of-Concept Setup



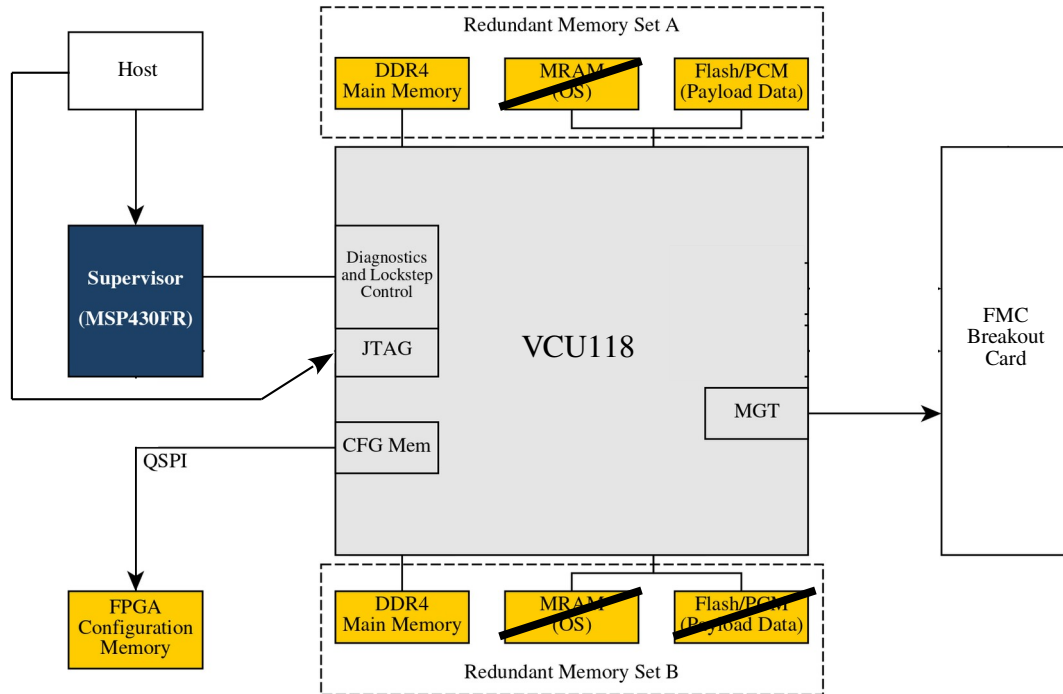
Current Proof-of-Concept Setup

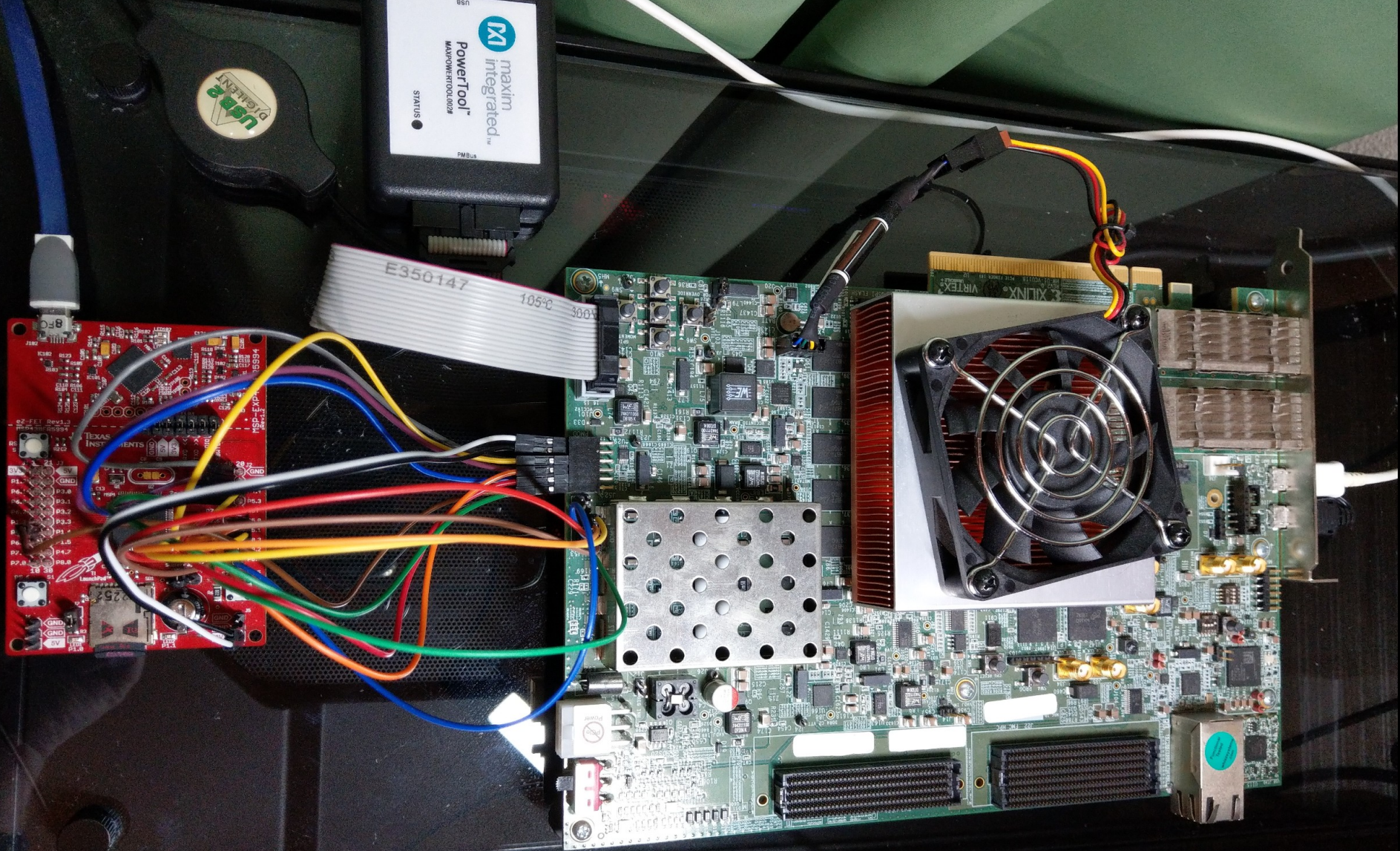


Current Proof-of-Concept Setup

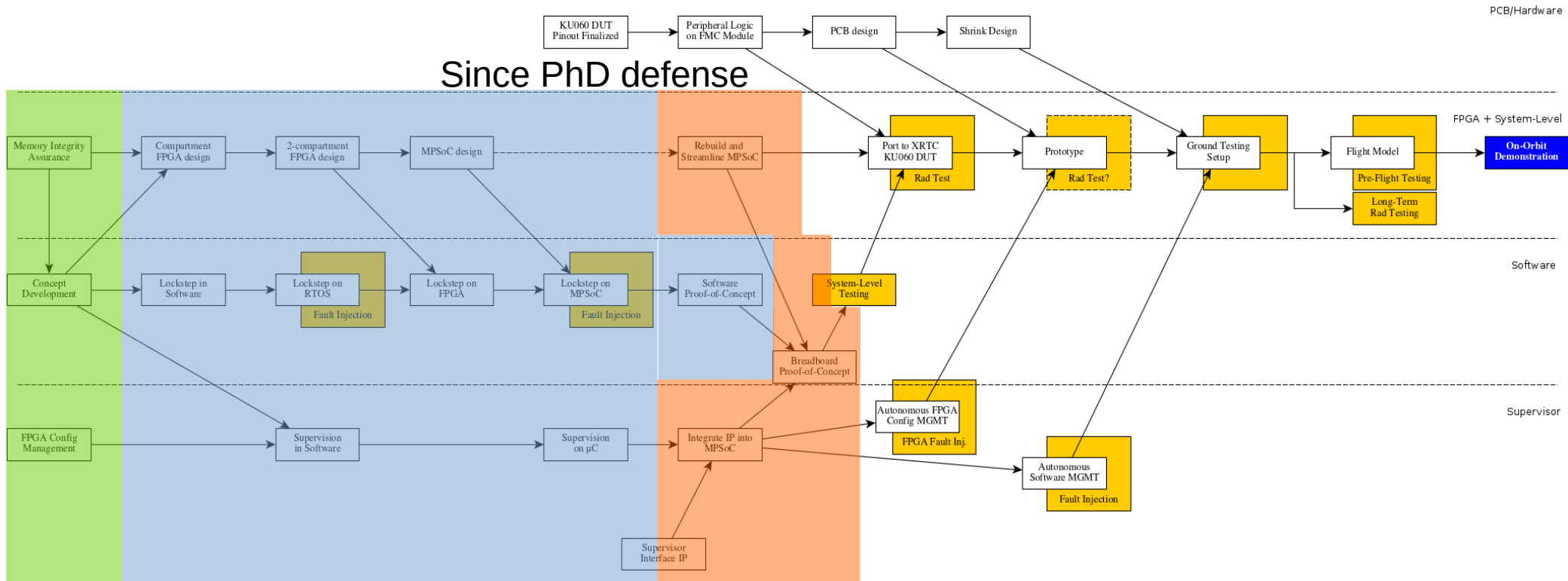


Current Proof-of-Concept Setup

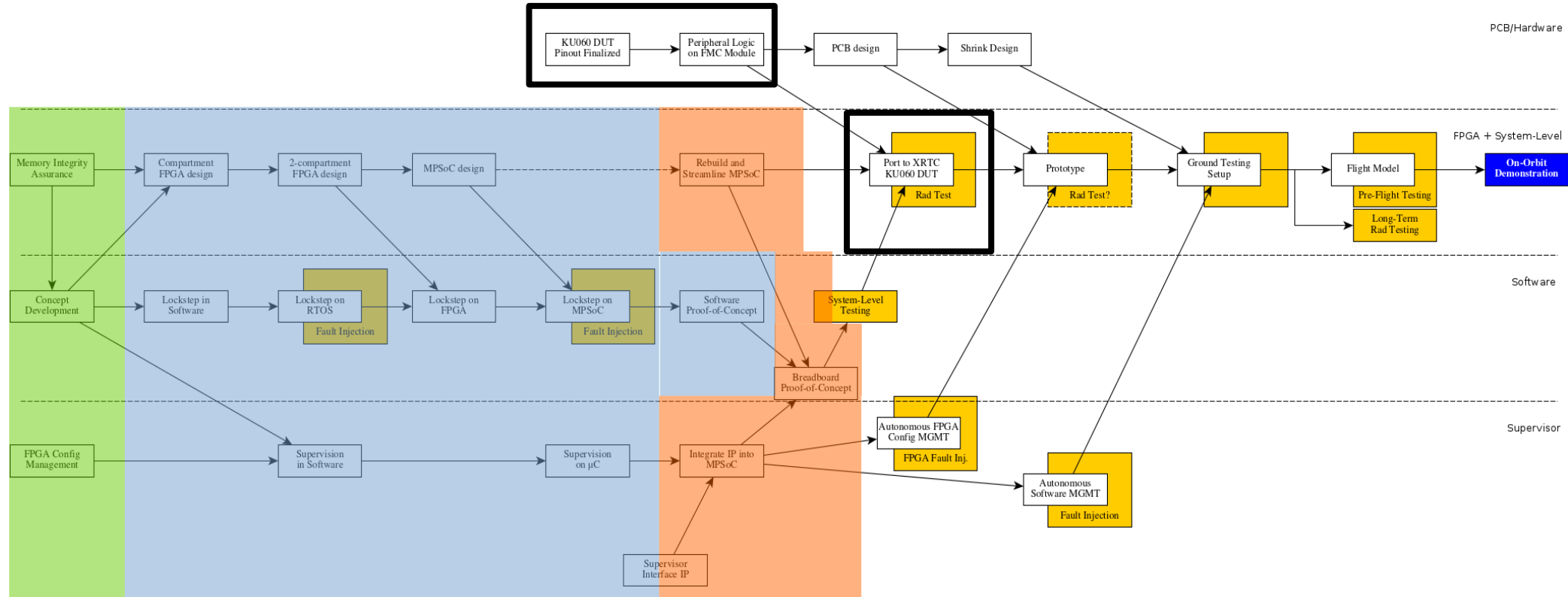




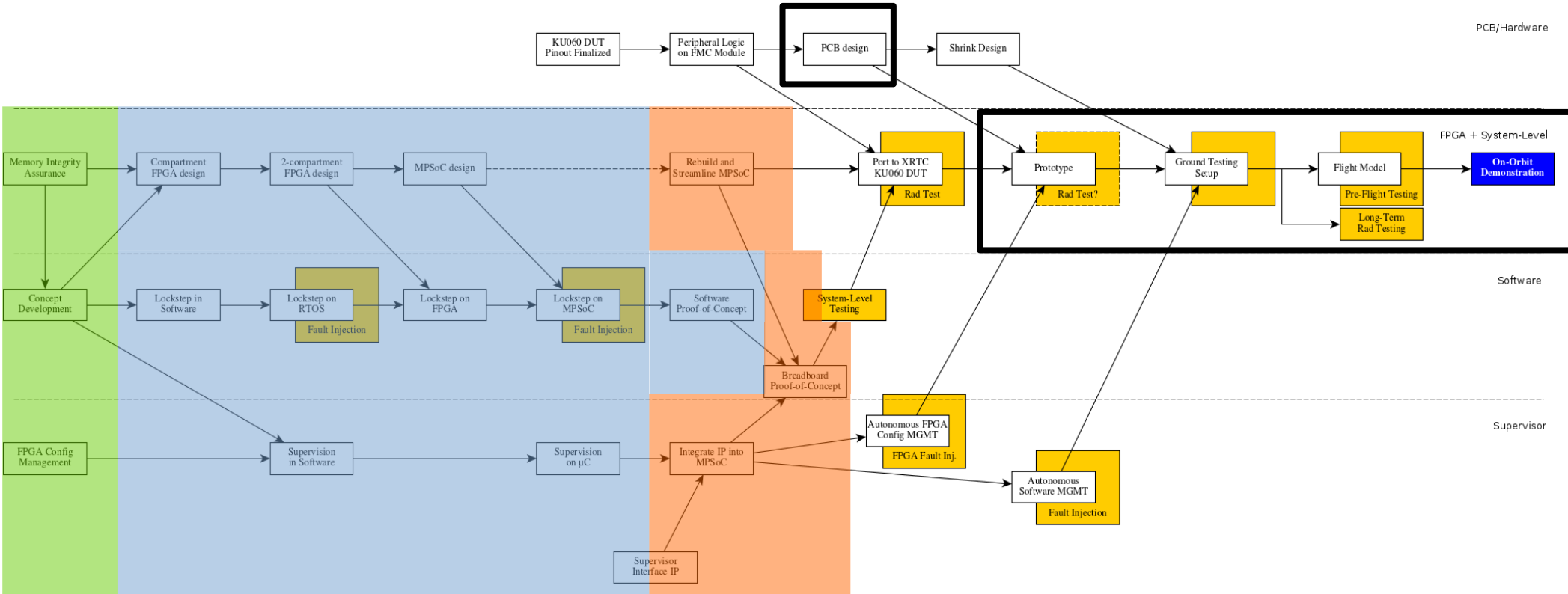
Development Roadmap



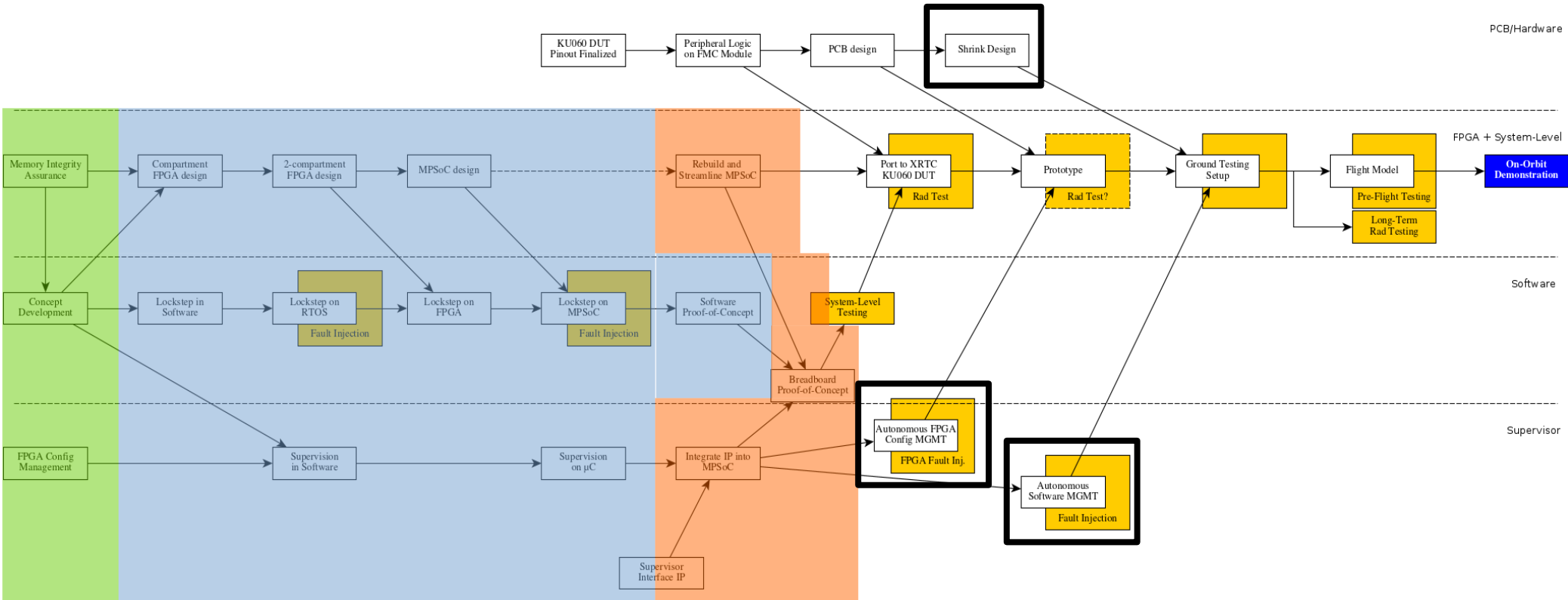
Development Roadmap



Development Roadmap



Development Roadmap



What Comes Next?

- **Find a position where I can properly build and test a prototype**
- Port design to final KU60DuT card
- Make breakout FMC modules for VCU118 setup KU60DuT
- Radiation testing → waiting for KU60DuT
- Construct prototype
- Find people & resources to take this to maturity to commercialize